GPU Kernel Compilation in Polygeist/MLIR: **Representing GPU Computation**

Ivan R. Ivanov[1,2]

Alex Zinenko[3], Jens Domke[2], Endo Toshio[1], Johannes Doerfert[4], William S. Moses[5]

1. Tokyo Institute Of Technology
2. RIKEN R-CCS
3. Google
4. Lawrence Livermore National Laboratory
5. UIUC

Problem Aim and Contributions

Writing high performance portable CUDA code is **difficult**

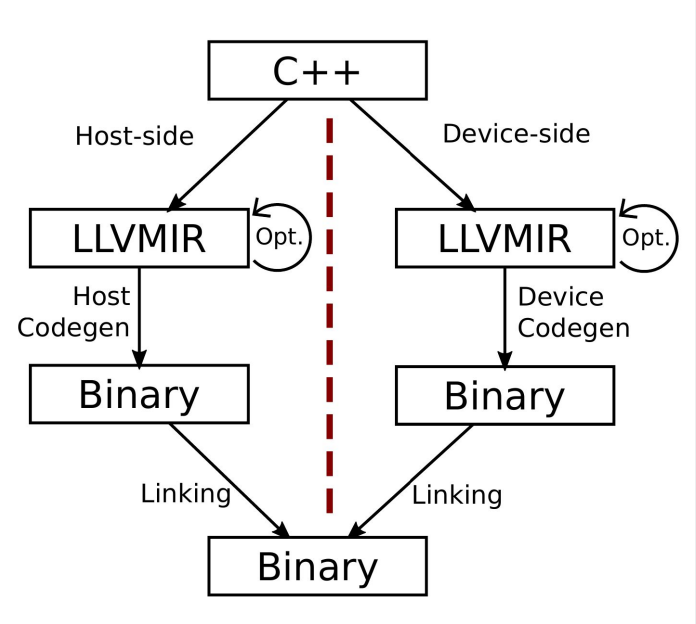
Write simple CUDA code once - let the compiler optimize and tune it for the target architecture (even AMD GPUs!)

How?

- **Represent parallel GPU computation in MLIR** (Extend Polygeist)
- Optimizations for **maximizing target hardware utilization**
- **CUDA to AMDGPU translation**

Traditional compilers

Device and Host side code split at the start of the pipeline (Example: clang)



The information about the parallelism is hidden behind a library call

```
; Host side
declare @foo
define void @main() {
  ...
  call @cudaLaunchKernel(... , @foo)
}
```

```
; Device side
define void @foo(%a, %b, %c, %d, %i) {
  %s = call @sqrt(%d) ; Executed N times
  c[%i] = a[%i] + b[%i] + %s
}
```

The single thread work, where is the parallelism?

Optimization Friendly Parallel Representation in MLIR

GPU computation with **parallel semantics** and **context**

device region

grid

shared memory

block

synchronisation

```
1  func @launch(%h_out : memref<?xf32>,
2               %h_in  : memref<?xf32>, %n : i64) {
3    // Host code
4    %d_out = gpu.alloc ...
5    %d_in = gpu.alloc ...
6    gpu.memcpy %d_in %h_in
7    polygeist.gpu_region {
8      // Device code
9      parallel.for (%bx, %by, %bz) = (0, 0, 0)
10                to (grid.x, grid.y, grid.z) {
11       %shared_val = memref.alloca : memref<f32>
12       parallel.for (%tx, %ty, %tz) = (0, 0, 0)
13                  to (blk.x, blk.y, blk.z) {
14         if %tx == 0 {
15           %sum = func.call @sum(%d_in, %n)
16           memref.store %sum, %shared_val[] : memref<f32>
17         }
18         polygeist.barrier(%tx, %ty, %tz)
19         %tid = %bx + blk.x * %tx
20         if %tid < %n {
21           %res = ...
22           memref.store %res, %d_out[%tid] : memref<?xf32>
23         }
24       }
25     }
26   }
27   gpu.memcpy %h_out %d_out
28 }
```

[1] High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. (PPoPP '23).

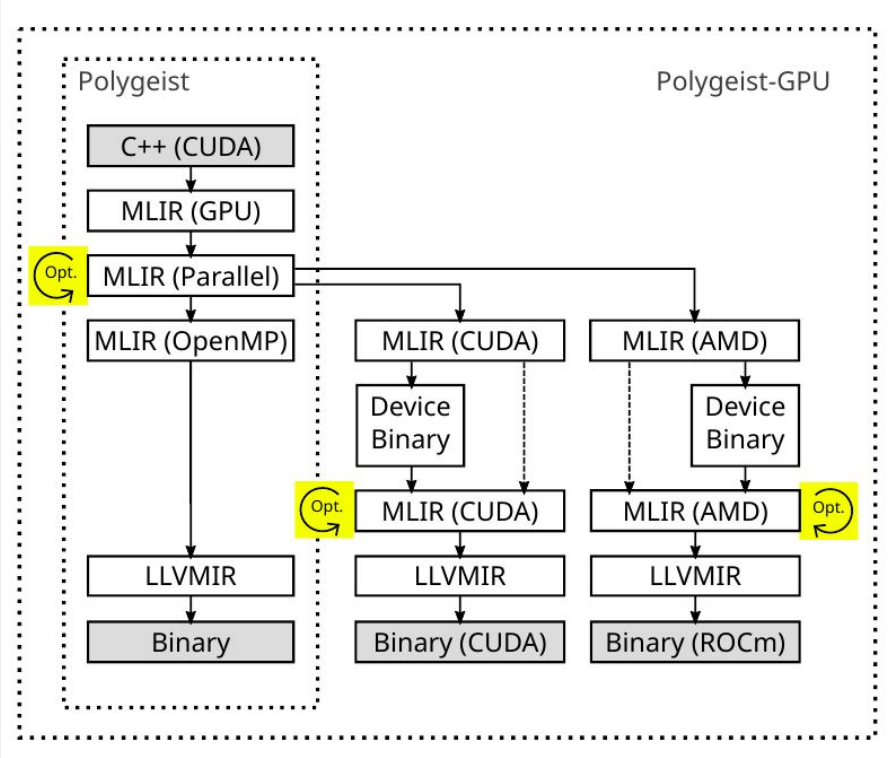
Support for GPU Compilation in Polygeist: The pipeline

Added

- Parallel -> Generic GPU

Adapted from upstream MLIR

- Generic GPU -> CUDA
- Generic GPU -> AMDGPU



Recursive Unroll and Interleave

A generic version of thread coarsening

```
1  parallel %i = from 0 to 16 {
2    A(%i)
3    B(%i)
4    C(%i)
5  }
```

```
1  parallel %i = from 0 to 8 {
2    %i_0 = %i * 2
3    %i_1 = %i * 2 + 1
4    A(%i_0)
5    A(%i_1)
6    B(%i_0)
7    B(%i_1)
8    C(%i_0)
9    C(%i_1)
10 }
```

Recursive Unroll Interleave and Jam

A generic version of thread coarsening

We can apply it to blocks!

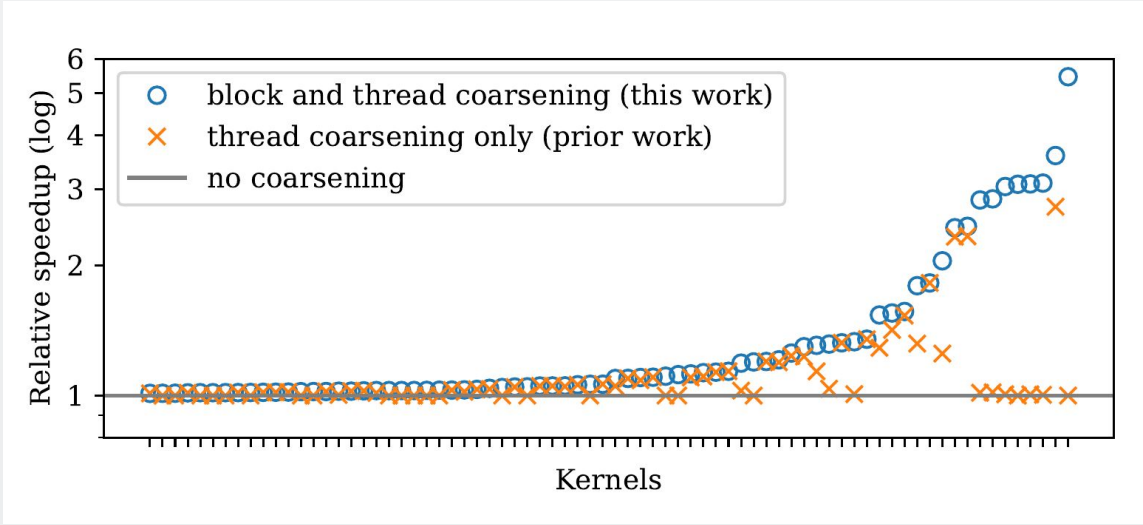
```
1  parallel %block = 0 to %n {
2    %shmem = alloca()
3    parallel %thread = 0 to 1024 {
4      A(%block, %thread)
5      B(%block, %thread)
6    }
7  }
```

```
1  parallel %block = 0 to (%n / 2) {
2    %block_0 = %block * 2
3    %block_1 = %block * 2 + 1
4    %shmem_0 = alloca()
5    %shmem_1 = alloca()
6    parallel %thread = 0 to 1024 {
7      A(%block_0, %thread)
8      A(%block_1, %thread)
9      B(%block_0, %thread)
10     B(%block_1, %thread)
11   }
12 }
```

We can even combine the two

Evaluation

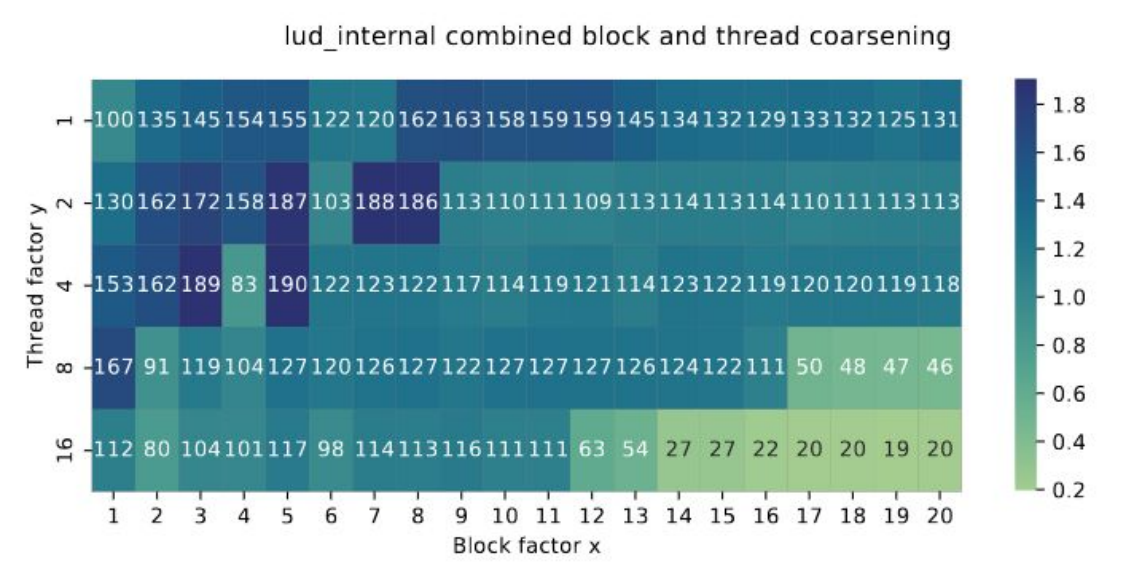
On the supported HeCBench and Rodinia CUDA kernels



(only the ones with speedup > 1%)

Some interesting results…

Finer granularity of block coarsening needed to maximise performance



Unroll Factor of 5? 3?

Conclusion

Nice representation of the computation

Better (and easier) parallel optimizations